Memristive Logic: A Framework for Evaluation and Comparison

John Reuben, Rotem Ben-Hur, Nimrod Wald, Nishil Talati, Ameer Haj Ali,
Pierre-Emmanuel Gaillardon*, and Shahar Kvatinsky
Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion - Israel Institute of Technology, Haifa, Israel 3200003
Email: johnreuben@technion.ac.il,{rotembenhur, nimrodw, nishil.t, ameerh}@campus.technion.ac.il, shahar@ee.technion.ac.il
*University of Utah, Salt Lake City, UT, USA Email: pierre-emmanuel.gaillardon@utah.edu

***Abstract*—Memristors have extended their influence beyond memory to logic and in-memory computing. Memristive logic design, the methodology of designing logic circuits using memristors, is an emerging concept whose growth is fueled by the quest for energy efficient computing systems. As a result, many memristive logic families have evolved with different attributes, and a mature comparison among them is needed to judge their merit. This paper presents a framework for comparing logic families by classifying them on the basis of fundamental properties such as statefulness, proximity (from the memory array), and flexibility of computation. We propose metrics to compare memristive logic families using analytic expressions for performance (latency), energy efficiency, and area. Then, we provide guidelines for a holistic comparison of logic families and set the stage for the evolution of new logic families.**

I. Introduction

Memristive technologies are promising for nonvolatile memory (NVM) design because of their high speed, low power, scalability, data retention, endurance, and compatibility with conventional CMOS in terms of fabrication and operating voltages [1]. A memristive device (or a 'memristor' in short) is a two-terminal device whose resistance is determined by an internal state, which can be varied by the application of a voltage/current. The capability to toggle resistance (between a Low Resistance State - LRS, and a High Resistance State - HRS) in response to voltage/current is perhaps the most desirable property of memristors, extending their use from memory to computing. Logic design using memristors is the field of designing logic circuits that use memristors as the primary computing device. The emergence of the memristor as an NVM device which can compute, at a time when modern computers are facing the memory wall problem, has set the stage for memristors to be efficiently deployed for in-memory computing. The memory wall problem has two facets: the mismatch in the performance of processor and memory, and the energy for memory access, which is growing exponentially along the memory hierarchy (from cache to off–chip DRAM) [2]. There has been a continuous effort to move processing closer to where data resides, to alleviate the memory wall problem. Memristive logic can integrate processing and storage seamlessly, an attribute which, if exploited well, can be a promising solution to scale the memory wall. Consequently, many memristive logic families have emerged with different characteristics and capabilities, with the goal of exploiting memristors for logic and for in-memory computing.

A memristive logic family defines the manner of voltage application and connection pattern between circuit elements, including memristors, to compute a certain primitive logic (AND, OR, NOR, NOT, XOR, *etc.*). Complex Boolean functions can be executed using these primitive gates as building blocks. Some memristive logic families are listed in Table I. Although this is only a partial list, it represents the different types of logic families proposed in recent years, and will be used to facilitate our comparison. Despite their fundamentally different characteristics and capabilities, most of these logic families share the goal of trying to solve the memory wall problem by computing in memory.

However, there is no clear method for classifying memristive logic families according to their fundamental properties. This has lead to unfair comparison of memristive logic families in recent research literature. Furthermore, as we show in this paper, not all of the proposed memristive logic families can perform logic within memory, and if they can, comparing their attributes to other logic families (memristive, as well as non-memristive) is not straightforward.

This paper establishes a framework for classifying and comparing different logic families from both circuit and system point of view. First, we classify memristive logic families by certain fundamental properties. Then, we propose various metrics to compare them holistically. We limit our discussion to memristive Boolean logic, although other types of memristive logic, such as threshold logic and neural logic, exist as well [3]. We classify memristive logic families into three categories. First, we classify them according to the consistency of representing data throughout the computation using one physical quantity. We call this category as *statefulness* (Section II-A). The second category is the location where the computation is performed and the hardware that participates in the computation. We call this category the *proximity of computation* (Section II-B). The third category is the *flexibility* (Section II-C) to compute different logic operations. After categorizing the memristive logic families, we present the desirable characteristics of a logic enabled memory and the peripheral circuit around it (Section III). We then propose the evaluation metrics to compare different

TABLE I
DIFFERENT MEMRISTIVE LOGIC FAMILIES

Acronym (logic family name)	Reference
IMPLY (material implication)	[4], [5]
MAGIC (Memristor Aided loGIC)	[6], [7]
FBLC (Fast Boolean Logic Circuit)	[8], [9]
MAJ (Majority based logic)	[10]
IMEC (In MEmory Computing)	[11]
MRL (Memristor Ratioed Logic)	[12]
MAD (Memristor As Driver)	[13]
PIPM (Parallel Input Processing Memristor)	[14],[15]
PINATUBO	[16]
Akers (memristive Akers logic array)	[17]

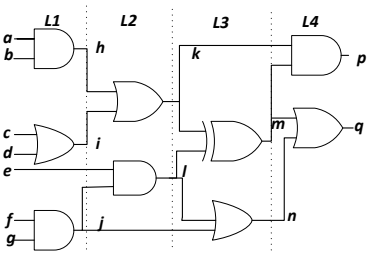

Fig. 1. In CMOS logic, inputs (a-g), outputs (p, q) and intermediate values (h, i, j, k, l, m, n) are represented as voltages; in memristive logic, inputs, outputs and intermediate values are represented as either voltages or resistances.

logic families (Section III). We present three metrics for evaluation: performance in terms of latency (Section IV), energy efficiency (Section V), and area efficiency (Section VI). Then, we show how to estimate each of these metrics in a generic manner (not specific to a particular logic family or technology implementation) based on the proposed classes of the logic family. Using this framework, we will set the foundation for a fair comparison among different memristive logic families and also with CMOS based computation.

II. CLASSIFICATION OF MEMRISTIVE LOGIC FAMILIES

Memristive logic differs fundamentally from conventional CMOS logic. In CMOS logic, there is only a single logic state variable, *i.e.,* voltage. The input data is represented as voltage and is processed as voltage throughout the computation (including in all of the intermediate stages), and is finally also represented as voltage at the output, as illustrated in Fig. 1. Furthermore, the state variable is regenerated throughout the computation by the CMOS gates. This seamless flow is disrupted in memristive logic because the internal state of memristors governs their resistance, introducing resistance, in addition to voltage, as a logic state variable for computation. The computation techniques in memristive logic usually rely on these two state variables through Ohm's law. The interaction between resistive states of the memristors and voltages are used to produce different dynamic behaviors that enable the circuit to execute the desired logic operation.

This fundamental difference has manifested in two different types of memristive logic families: one where resistance is the only logic state variable for representing inputs, output, and intermediate results of computation, and the other where voltage and resistance are both used as part of the computation. This forms the basis for our first classification category, *i.e.,* statefulness. The second classification category stems from the fact that memristive logic can integrate processing and storage; hence, it is necessary to determine how tightly a logic family can couple processing and storage. The third classification category, flexibility, is based on the extent to which a memristive logic circuit can be reconfigured to execute different logic functions using the same computing fabric.

A. Statefulness

A memristive logic family is said to be stateful if the Boolean variable is represented only as the state of the memristor (*i.e.,* resistance) and computation is performed by manipulating this state [18]. In other words, inputs are represented as resistance and the output(s) after computation is (are) also stored as resistance of the memristor. IMPLY and MAGIC are examples of stateful memristive logic families. In some logic families, the input state variable is a voltage and the output is stored as resistance (*e.g.*, MAJ), which is not stateful. Similarly, there can be logic families where the input is represented as resistance and the output after logic computation is sensed as voltage (*e.g.,* PINATUBO and Akers). Such families are not stateful either. In the MRL family, the input and output state variables are both voltages (as in CMOS logic) and memristors are used as voltage dividers only to determine the output.

Statefulness is a fundamental classification because the statefulness of a logic family has far reaching effects on its compatibility with other units, such as CMOS-based circuits and memristive memory cells. If the circuits are incompatible, state conversion (from resistance to voltage or vice versa) of consecutive logic operations will be required, influencing performance, power, and area. Consequently, statefulness is a desired characteristic for computation within memristive memory since computation is performed using the same logic state variables as represented in the memory cells and, as a result, no conversion is required. On the other hand, non-stateful families benefit from better integration with CMOS. Additional stateful logic families include MAD and FBLC; additional non-stateful families include PIPM and IMEC.

B. Proximity of Computation

From the early '90s, many researchers have been searching for ways to scale the memory wall by bridging the gap between where data is stored and where it is processed. Early researchers used the term processing-in-memory (PIM) to refer to the effort to move the processing closer to where data resides [19]. The term processing-in-memory broadly referred to processing in the memory using computing units placed in the memory chip. For example, in [19], processing units like ALU were placed in the periphery of the memory array in the memory chip. Later researchers used the term near-memory computing or near data processing to refer to the same effort and they exploited 3D stacking of DRAM dies over logic die to compute near memory [20].

The terms processing-in-memory and near-memory computing meant the same effort towards the goal of processing data without requiring costly off-chip data transfer. The term processing-in-memory can be misconstrued because it has a broader meaning and does not specify whether processing is done in the memory cells inside the memory array or in an area outside the memory array in the memory chip. It is a system-level definition which is agnostic to the underlying device/circuit technology. Traditionally, memory cells could only store data. Emerging NVM technologies, however, can compute as well as store. In light of this, we need to re-define 'near memory' computing and 'processing in-memory' more precisely.

Our terminology is ruled by the data movement requirement of the memristive logic family, since computation is dominated by data movement and not by the computation itself [2]. Therefore, we re-define PIM and near-memory computing based on the location of data with respect to the memory array, during computation, $i.e$, the proximity of computation. We define the memory array as a regular array of memory cells to store data, replicated in two dimensions, the wordline and the bitline, and not including its auxiliary circuit. We re-define processing-in-memory as 'in-memory computing' and define it as the computing model in which data resides only within the memory array during the entire computation.

We re-define 'near-memory computing' as the computing model which requires data movement to the auxiliary circuit ($e.g.$, for state conversion) during the course of computation, even if some (or most) of the computation is carried out by the memory cells. Note that according to this terminology, the recently proposed "in-memory computing" techniques for SRAM [21] and DRAM [22] that use sense amplifiers to sense charge sharing among memory cells are actually near-memory computing. When the data is moved out of the memory array and the entire computation is performed in a dedicated processing area outside the memory array, the proximity is defined as 'out-of-memory computing'. Hence, the memory array is the point of reference in our definition. In out-of memory computing, computation may be performed in an area outside the memory array or even in another die ($e.g.$, a logic die beneath a DRAM die as in the hybrid memory cube) or in another chip (as in conventional von Neumann machines).

Consider a simple Boolean logic function which, due to data dependencies, has to be executed as four logic levels, as shown in Fig. 1. A logic family implements in-memory computing if it does not require any data to be read out of the memory array untill all the four levels are computed, as shown in Fig. 2(a). In MAJ, two out of the three inputs to a MAJ gate are voltages, resulting in a need for state conversion. Hence, MAJ requires that data (intermediate values of computation) be moved out of the array between every logic level and written to the array as inputs of the next stage (Fig. 2 (b)). In IMEC, each memory subarray implements minterms and two subarrays together implement logic in SOP form, requiring data movement to implement a two-level logic (Fig. 2 (c)). In PIPM, the data from the memory array has to be passed to a summing amplifier (in CMOS), resulting in data movement for every logic gate evaluation, as shown in Fig. 2 (d).

C. Flexibility

A memristive logic family is said to be flexible if a variety of operations can be executed using the same computing elements. To achieve flexibility, a logic family has to provide a basic operation (or a set thereof) which is functionally complete, and allow different control signal sequences to result in different outcomes. Some of the logic families are similar to ASIC, where the functionality of each computing element is determined prior to the fabrication process. Hence, they can perform a fixed function (or set of functions). For example, MRL, MAD and FBLC are all non-flexible since each design yields specific logic operations. On the other hand, IMPLY, MAGIC, MAJ and IMEC are all flexible, $i.e.$, different computations can be executed using the same computing units at different execution times, and therefore the functionality can be dynamically chosen during runtime.

All flexible families require a controller that conducts the execution of the desired program using the adjustable computing elements and synchronizes the sequence of basic logic operations supported by the family. Some sort of compiler or logic synthesis tool is necessary in order to generate an efficient sequence of basic logic operations to realize a desired function. Using an inadequate synthesis tool can lead to an inefficient logic implementation in terms of performance and/or power, while the proper use of it can result in a cost-effective design.

Non-flexible families can become programmable in a similar manner to a general-purpose CPU, where the designed fixed-functions are sufficient to perform any required task and construct a desired datapath. Programmable non-flexible families can compute any desired operation, but cannot be used in or near memory since they cannot be made compatible with the memory array.

III. Logic Enabled Memory and Evaluation Metrics for Memristive Logic Families

The conventional memory used for storage needs a simple controller and the associated peripheral circuitry that supports read/write operations. The in-memory and near-memory computation models presented in section II-B require a memory which can support logic operations, a logic enabled memory.

A. Logic Enabled Memory

A logic enabled memory should *enable* logic with minimal modifications to the memory structure. As defined, the memory array has to be a uniform replication of cells and the uniformity should not be disturbed in the process of enabling logic in it. Each cell may have a selector (*e.g.,* a transistor or a diode) in addition to the memory device, but nevertheless uniform. Similarly, the peripheral circuit and controller should be modified carefully to enable logic, while still supporting the memory operations. To support logic operations in a memory array, the major requirements are:

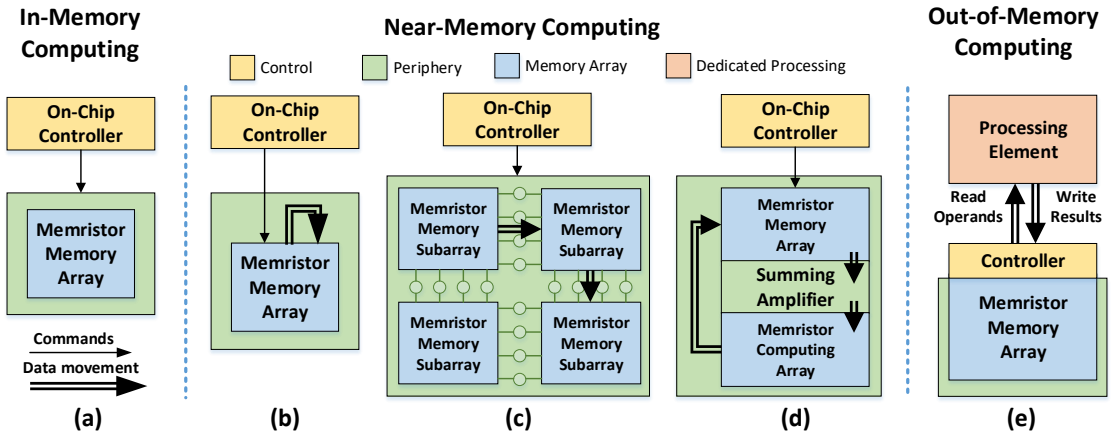

Fig. 2. Illustration of data movement in systems with different proximity of computing. (a) Data does not move out of the memory array during entire computation in in-memory computing (*e.g.*, MAGIC and IMPLY); data moves out of the memory array in near-memory computing (b) between consecutive logic levels (*e.g.*, MAJ); (c) for computing each Sum-of-Products (*e.g.*, IMEC); (d) for computing each logic gate (*e.g.*, PIPM); (e) data is read out of the memory, processed in dedicated elements and results are written back in out-of-memory computing.

1) The topology of circuits constructed in a memristive logic family must be compatible with the memory array.
2) The peripheral circuitry and controller (together called *auxiliary circuit*) have to support the logic family and therefore be augmented with extra circuitry (*e.g.*, additional voltage sources, different cell selection schemes, state conversion, *etc.*).

1) Memory array compatibility: A memristive logic family cannot be fully exploited for in-memory/near-memory computation if its basic logic gate cannot be realized in the array structure of the memory. Many memristive logic families are array compatible (*e.g.*, IMPLY, MAGIC, IMEC, MAJ, and PINATUBO). Some of those that are not can be modified to have some compatibility. For example, Akers can be integrated in a memory array with modified memory cells that have four transistors and two memristors in each cell. A different modification can be made to the periphery circuits to add compatibility with memory arrays. For example, the original PIPM [14] was not array compatible, and was made array compatible by adding extra peripheral circuitry [15]. FBLC, as presented in [8], requires special arrays with disabled memristors, which cannot co-exist with the normal array used for storage, and hence it is said to be array incompatible.

2) Peripheral Circuitry: Peripheral circuitry around the memory array has not been proposed in many logic families (*e.g.*, MAGIC, IMPLY and MAJ), and the lack of this circuitry blurs the system view. The peripheral circuitry for in-memory and near-memory logic families should enhance the capability of the peripheral circuitry used for memory, and the enhancements should include:

- The ability to select multiple rows/columns to enable more parallel execution of logic operations.
- The ability to apply additional distinct voltages beyond the voltages used for memory operation (V_{READ}, V_{SET}, *etc.*).
- Additional devices required to support the logical operations. For example, an additional resistor in each row in IMPLY.

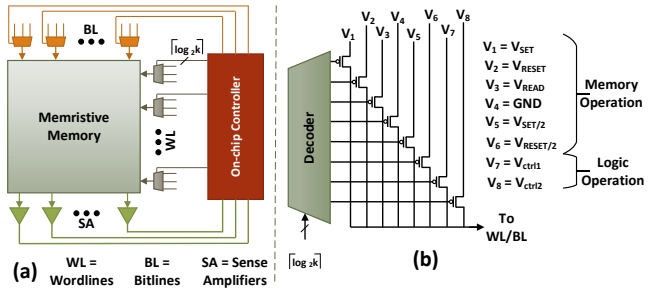

Fig. 3. (a) Peripheral circuit around memory. (b) Structure of an analog mux.

To facilitate comparison among different families, we need a generic peripheral circuitry with this enhanced capability. To this end, we adopt the circuitry proposed in [23]. As shown in Fig. 3(a), the wordlines and bitlines are fed through analog muxes, and sense amplifiers at the end of the bitlines are used to read the data from memory. The analog mux is used to select the voltage that needs to be applied at every clock cycle based on the 'voltage select' signal received from the controller. The controller is a Finite State Machine (FSM) that orchestrates the computation. The analog mux has a $\lceil \log_2^k \rceil{:}k$ decoder which selects one of the k voltages and pulls up the wordline to the selected voltage (Fig. 3(b)). The voltages ($V_{SET}, ..., V_{RESET/2}$) are the basic voltages required for memory operations and are common to all logic families. We assume a memristor with asymmetric switching and use $V_{SET/2}$ and $V_{RESET/2}$ as half-select voltages [6]. Voltages V_{cntrli} are the extra control voltages required to execute logic and they are specific to a logic family. For example, IMPLY requires two control voltages, while MAGIC requires only a single control voltage. Some logic families, such as MAJ, do not require these extra control voltages since they can perform computation using the voltages used for memory operations. The number of distinct control voltages used in a logic family is an important parameter since it determines

the size/complexity of the peripheral circuitry. Each control voltage makes the size of each multiplexer larger. In addition to this basic circuit, some logic families require extra hardware in the peripheral circuit. For example, PIPM requires summing amplifiers in the peripheral circuit.

B. Evaluation Metrics

Energy to compute a task and time taken to compute a task (latency) are the fundamental performance metrics or figures of merit of any computing model. It is not possible to define a baseline operation to compare the latency and energy efficiency of logic families, since each family is different in the logic primitives it supports and the degree of parallelism it offers. Consequently, a good memristive logic family is one which consumes the least energy and takes the least time to compute a common benchmarking task. Additionally, the area to compute a task must also be considered. This needs a new perspective when we consider in-memory and near-memory computing. In memristive logic, a sequence of control voltages are applied to execute combinational logic. The latency of an operation in a logic family depends on the switching time of the memristors in the memory array, as well as the latency of other tasks that are carried out between consecutive operations, like reading from and writing to the memory array (in the case of near memory computing). We discuss performance (latency per operation) of logic families in Section IV. During a logic operation, the on-chip controller commands the peripheral circuit to apply appropriate voltages, which switches the memristors, resulting in the desired output. Hence, an insight into energy per operation can be obtained by analyzing the energy consumption in each of these parts – the controller, peripheral circuit and memristors (in the memory array or elsewhere), during the operation. In section V, we discuss the energy efficiency of different logic families. The area efficiency of logic families is a measure of the area overhead to the basic memory to enable computation and is discussed in Section VI. We base our comparison on proximity of computation since proximity gives the structural view of the system and makes the comparison clear.

IV. Latency of Memristive Logic Families

The computational latency of in-memory and near-memory logic families depends on the frequency at which control signals are applied, which in turn depends on physical factors such as the memristor technology used, wire parasitics, array size *etc.* We define T_{clk} to be the clock period of the memory clock, and T_{write}, T_{read}, and T_{logic} to be the time to perform write, read, and logic operations, respectively. Depending on the memristive technology and logic family, values of these timing parameters could be one to several T_{clk}.

We define T_{sw} to be the switching time of a single memristor. Depending on the location of the memristor in the memory array, the switching time varies because of array parasitic effects. For example, the switching time will be the highest for the memristor farthest from the voltage driver due to IR drop across array interconnect [24]. Hence, T_{write} and T_{logic} must be sufficiently long to accommodate the charging/discharging time of the wordlines/bitlines and the worst case switching time of the memristor.

To compute complex logic functions, each memristive family executes a series of basic logic operations: NOR operations in the MAGIC family, for example, or IMPLY and FALSE operations in the IMPLY family. Depending on the parallelism offered by the logic family, in each step, one or more of these basic functions can be executed. We define the number of computational steps, NCS, as the number of steps for a particular operation where, in each step, a basic operation(s) of the logic family is (are) executed.

A. In-Memory Logic Families

For in-memory logic families, the latency is the time to compute a given operation and is

$$T_{compute} = NCS \cdot T_{logic}. \tag{1}$$

Note that, in addition to the difference in NCS among different memristive logic families, each logic family has its own T_{logic}, since the delay of the basic logic operation depends on the connection pattern among circuit elements and the voltage levels. For example, the NCS for a 1-bit full adder operation in a serial execution of IMPLY logic is 29 [4] and in the MAGIC family, it is 15 [6]. Hence, the latency for a 1-bit full-adder operation is $29{\cdot}T_{logic\_IMPLY}$ for IMPLY and $15{\cdot}T_{logic\_MAGIC}$ for MAGIC.

B. Near-Memory Logic Families

In near-memory logic families, there is data movement in and out of the memory array during the computation, since the peripheral circuit is also partially involved in the computation. Sometimes data movement is required for logic state conversion (from resistance to voltage, or to any other conversion form) for the subsequent stages of computation. In MAJ, for example, the output of a logic level is stored as resistance, and is needed as voltage for the next logic level. In some logic families, data is processed in the peripheral circuit, and written back to memory array, if it is needed as resistance for the next stage of computation. The time for reading data out of the memory array, processing it, and writing back to the memory array during the course of computation must be added to (1). The read time depends on the delay in the memory array and the CMOS-based sensing circuit. For example, an RRAM memory fabricated by Panasonic has read-out time of 25 ns, while the bipolar switching time is only 10 ns [25], implying that the read-out time cannot be ignored when determining the time to compute. The latency of a near-memory logic family is therefore

$$\begin{aligned} T_{compute} =& NCS \cdot T_{logic} + N_{read}(NCS) \cdot T_{read} + \\ & N_P(NCS) \cdot T_P + N_{write}(NCS) \cdot T_{write}, \end{aligned} \tag{2}$$

where N_{read}, N_{write}, and N_P are, respectively, the number of reads, writes, and processing (in the auxiliary circuit), and are dependent in NCS (the exact dependency varies among different memristive logic families). T_{read}, T_{write}, and T_P are, respectively, the time to read, write and process the data (if

needed) using the auxiliary circuit. In MAJ, for example, data is written back during logic phase as the input of the next logic stage, thus requiring no overhead for the write operation ($i.e.$, T_{write}=0). Furthermore, T_P can be considered as the time to convert the read voltage to the corresponding control signals to be applied at the appropriate wordline/bitline, $i.e.$, the latency through the controller. As discussed in Section II-B, N_{read} and N_{write} will increase as we move away from in-memory computing, resulting in increased time to compute. This is why NCS and T_{logic} alone cannot be used to determine and compare the latency of memristive logic families.

C. Out-of-Memory Logic Families

Out-of-memory computation is a conventional von Neumann computation, even if the (external to the memory array) processing units are made of memristors. The latency for out-of-memory computation is therefore

$$T_{oMem} = T_{read} + T_{compute} + T_{writeback}. \tag{3}$$

Each parameter of the latency depends on the computation model, memory hierarchy, technology, application, *etc.* Numerous examples of computation models that fall under this category exist including MRL, FBLC, computing using Hybrid Memory Cube (HMC), conventional von Neumann architecture, $etc.$ For example, modern computers are designed using memory hierarchy (three to four levels of cache, main memory, and storage) and CPUs. To perform computation, data has to be fetched from respective memory using a bandwidth-limited bus (T_{read}), processed in CPU ($T_{compute}$), and stored back ($T_{writeback}$) to the memory, which accounts for all the timing parameters throughout the execution of an application.

V. Energy Efficiency of Memristive Logic Families

To determine energy of in-memory and near-memory families, all of the units that participate in the computation must be considered. Hence, the energy of the memory array, the peripheral circuit and the controller needs to be determined. For out-of-memory families, the energy depends on data movement and the processing circuit outside the memory array. In this section, we give expressions for energy per operation. The exact definition of 'an operation' varies between different logic families and therefore a fair comparison must be on equivalent computational tasks.

A. Energy of In-Memory Logic Families

1) Energy consumed in the memory array: We divide the energy consumed in the array into static and dynamic contributions, *i.e.,*

$$E_{array} = E_{dynamic} + E_{static}. \tag{4}$$

During an in-memory logic operation, dynamic energy $E_{dynamic}$ is consumed by the memristors that switch during the operation and by the charging/discharging of the array wire capacitance. Assume the average switching energy of a single memristor is E_{sw}, and the collective array capacitor charging/discharging energy is E_{cap}, then the dynamic energy would be $E_{dynamic} = E_{cap} + n \cdot E_{sw}$, where n is the number of memristors that switch during the operation.

Currents flow through memristors that do not switch during the operation add static energy to the operation. We divide these memristors into memristors that are part of the computation but do not switch, and memristors which are not part of the computation. In memristors not involved in the computation, there are sneak path currents which exist in passive (selectorless) arrays and leakage current in arrays with selectors (*e.g.,* transistor leakage current in 1T1R). We collectively call the energy dissipation in all the memristors that do not switch as $E_{leakage}$. Apart from this, there is a residual static energy $E_{residue}$, which dissipates in the switching memristors after they switch. For example, since T_{logic} accommodates the worst case switching time (Section IV), after an output memristor switches, it can still consume energy since the control voltage is applied for a longer period to accommodate the worst case switching scenario. The total static energy in the array is therefore $E_{static} = E_{leakage} + E_{residue}$.

2) Energy consumed in the peripheral circuit: As discussed in Section III-A2, the peripheral circuit is responsible for applying the control voltages during the computation. Energy is consumed in the wordline/bitline multiplexers (which often consist of decoders and pass-transistors), voltage regulators, $etc.$ Sense amplifiers do not participate in the computation and therefore can be power gated and consume no energy during computation in an in-memory logic family.

3) Energy consumed in the controller circuit: A good estimate of the energy consumed in the controller circuit can be the number of states in its FSM, which depends on the number of computational steps NCS, and therefore $E_{controller} \propto NCS$. Additionally, the energy to fetch the control/program must be included in the energy consumed in the controller. In summary, the energy per operation of an in-memory logic family is

$$E_{op}^{IM} = E_{array} + E_{peripheral} + E_{controller}. \tag{5}$$

B. Energy of Near-Memory Logic Families

1) Energy consumed in the memory array: Since in-memory and near-memory logic families have the same array structure, the energy consumed in the memory array can be estimated in a similar manner as in Section V-A1. If the memory array has a different regular structure (for example, as in Akers [17]), the analysis should be done accordingly.

2) Energy consumed in the peripheral circuit: The energy consumed in the peripheral circuit must include, in addition to the energy discussed in section V-A2, the energy consumed due to data movement in the peripheral circuit of the memory array, and the energy to compute in the periphery. Due to the diverse working principles exploited by near-memory logic families, the energy consumed in the periphery is different for each. The most common way to convert the data in the memory array into voltage is by memory read operation using sense amplifiers (SA). The energy consumed while reading

and writing is

$$E_{read} = E_{SA} \cdot \sum_{i=1}^{N_{read}} N_b(i);\ E_{write} = E_w \cdot \sum_{i=1}^{N_{write}} N_b(i), \quad (6)$$

where E_{SA} is the energy consumed for reading a single bit of data using SA, E_w is the energy for writing a single bit of data into the memory and $N_b(i)$ is the number of bits read/written during a particular read/write operation.

The total energy dissipated in the peripheral circuit of a near-memory logic family is

$$E^{NM}_{peripheral} = E_{peripheral} + E_{read} + N_P \cdot E_P + E_{write}, \quad (7)$$

where $E_{peripheral}$ is the energy dissipated only due to application of control voltages in the peripheral circuit (as in in-memory logic families), E_P is the energy of processing within the periphery (if needed), and N_P is the number of such processing steps required during the course of computation.

3) Energy consumed in the controller circuit: The energy consumed in the controller circuit is be proportional to $(NCS + N_{read} + N_{write})$, since reading and writing data are also steps in the computation, $i.e.$

$$E^{NM}_{controller} \propto (NCS + N_{read} + N_{write}). \quad (8)$$

In summary, the energy per operation of near-memory logic family is

$$E^{NM}_{op} = E_{array} + E^{NM}_{peripheral} + E^{NM}_{controller}. \quad (9)$$

C. Energy of Out-of-Memory Logic Families

Generally, the energy for out-of-memory computation is

$$E_{oMem} = E_{read} + E_{compute} + E_{writeback}, \quad (10)$$

where E_{read} is the energy to read from memory, $E_{compute}$ is the actual execution energy in the processing unit, and $E_{writeback}$ is the energy for writing the result back to memory.

For FBLC logic family, E_{read} is the energy to read data from the memory, in which data is stored, to the computing array (special arrays with disabled memristors), wheraeas $E_{compute}$ is the energy to compute in the FBLC structure. The extreme end of the spectrum for out-of-memory computing is conventional von Neumann computing, where data is read out of memory and delivered to the processor for computing.

VI. Area evaluation of memristive logic families

When evaluating the area of a memristive logic family, one has to consider the area of all circuits participated in the logic functionality. For in-memory logic families, where computation is done by the memory cells, the area must include the memory cells participating in the computation, and of the changes made in the controller and peripheral circuitry in order to add computing ability to the memory cells. Changes to the periphery may be the addition of voltages to the analog multiplexers (and the addition of voltage drivers), as shown in Fig. 3 or any other augmentation of the circuits. Logic families that use a modified cell array to facilitate logic ($e.g.$, Akers) should include the area overhead of the modification as well when comparing to standard memory arrays.

In near-memory logic families, the computation is confined to the memory array and periphery. Any changes made to the controller and peripheral circuitry to accommodate computation ($e.g.$, summing amplifiers in PIPM or connection nodes in IMEC) need to be considered in the area evaluation as well. For out-of-memory logic families, data is read from the memory in the conventional way and processed in dedicated circuits. Thus, the area of the memory array and periphery does not contribute to the area of the computation. The only area that is counted for out-of-memory logic families is that of the dedicated circuits which perform the data processing. Examples of such dedicated circuits include arrays containing disabled memristors in FBLC, and hybrid memristor-CMOS AND/OR/NOT paths in MRL. The area to store the code of the computation task (program) must also be considered for all proximity classifications. Since the basic logic function is different among logic families, the memory area for code storage can vary substantially.

The area is therefore

$$A = A_{Cells} + A_{\Delta P\&C} + A_{PE} + A_{Code}. \quad (11)$$

A_{Cells} denotes the area of participating memory cells, and is non-zero only for in-memory and near-memory logic. Note that the inputs and outputs of the logic operation are excluded from this area since they are required in any type of computation. $A_{\Delta P\&C}$ describes the area added by changes to periphery and controller, and is relevant to in-memory, and to a greater extent, near-memory logic families. The area of dedicated processing circuits is A_{PE}, and is relevant only for out-of-memory logic families. A_{Code} is the area needed to store code for execution, and is required in all classes, but depends on the efficiency of the programming models and tools supporting the different logic families.

Another interesting metric, which applies only to in-memory and near-memory logic families, is the area utilized in the memory array for computation. As stated, complex logic functions are executed as a series of basic operations of a logic family. The intermediate results of the logical sequence are also stored as resistances of additional memristors, which cost additional area for the computation in the memory array. The memristors, where the intermediate results are stored, are called *functional memristors* [4],[6]. We define a new metric for comparison of different logic families, which we call *computing area utilization* to measure the overhead of the functional memristors involved in a certain operation. This metric not only depends on the logic family, but also on the algorithm used to execute various operations. Computing area utilization, η_{area}, for an operation is defined as

$$\eta_{area} = 1 - \frac{\#FM}{\#TM}, \quad (12)$$

where $\#FM$ is the number of functional memristors, and $\#TM$ is the number of memristors used in the computation. Since the number of input and output memristors is a property of the executed function, independent of the logic family, this metric gives a normalized comparison of the amount of functional memristors between families.

TABLE II
COMPARISON OF LOGIC FAMILIES OF DIFFERENT PROXIMITY TO MEMORY

	In-Memory	Near-Memory	Out-of memory*
Energy	$E_A + E_P + E_C$	$E_A + E^{NM}_{peripheral} + E^{NM}_{controller}$	$E_{read} + E_{compute} + E_{writeback}$
Latency	$NCS \cdot T_{logic}$	$NCS \cdot T_{logic} + N_{read}(NCS) \cdot T_{read} + N_P(NCS) \cdot T_P + N_{write}(NCS) \cdot T_{write}$	$T_{read} + T_{compute} + T_{writeback}$
Area	$A^{IM}_{Cells} + A^{IM}_{\Delta P\&C} + A_{Code}$	$A^{NM}_{Cells} + A^{NM}_{\Delta P\&C} + A_{Code}$	$A_{PE} + A_{Code}$

E_A, E_P, E_C are respectively the energy dissipated per operation in the memory array, peripheral circuit, and controller of an in-memory logic family while E^{NM}_P and E^{NM}_C are the energy dissipated per operation in the peripheral circuit and controller of a near-memory logic family.
* per application for out-of memory logic families

VII. CONCLUSIONS

Memristive logic can integrate processing and storage, a property which has reignited interest in the decades old concept of processing-in-memory. However, the conventional definitions of processing-in-memory and near-memory computing should be reevaluated in light of emerging memory technologies. We classified logic families based on their fundamental attributes and proposed metrics for the evaluation and characterization of memristive logic, which are summarized in Table II. While families classified to have the same 'proximity of computation' can be compared based on relatively small operations, the comparison of families with different 'proximity' has to take into account the full system scope. The comparison between in/near memory and out-of-memory computing can be done only at the application level, and not at the basic logic operation level. Memristive logic will herald a new era in computing, and if rightly exploited, has the potential to solve the von Neumann bottleneck.

ACKNOWLEDGMENT

This research was partially supported by Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), by the Viterbi Fellowship in the Technion Computer Engineering Center, and by EU COST Action IC1401.

REFERENCES

[1] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "The desired memristor for circuit designers," *IEEE Circuits Syst. Mag.*, vol. 13, no. 2, pp. 17–22, 2013.

[2] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Des. Test.*, vol. 34, no. 2, pp. 39–50, April 2017.

[3] G. S. Rose, J. Rajendran, H. Manem, R. Karri, and R. E. Pino, "Leveraging memristive systems in the construction of digital logic circuits," *Proc. IEEE*, vol. 100, no. 6, pp. 2033–2049, June 2012.

[4] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, Oct 2014.

[5] A. Raghuvanshi and M. Perkowski, "Logic synthesis and a generalized notation for memristor-realized material implication gates," in *International Conference on Computer-Aided Design*, Nov 2014, pp. 470–477.

[6] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *IEEE Trans. Nanotechnol.*, vol. 15, no. 4, pp. 635–650, July 2016.

[7] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC- Memristor-Aided Logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 11, pp. 895–899, Nov 2014.

[8] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Fast boolean logic mapped on memristor crossbar," in *International Conference on Computer Design*, Oct 2015, pp. 335–342.

[9] H. A. D. Nguyen, L. Xie, M. Taouil, S. Hamdioui, and K. Bertels, "Synthesizing HDL to memristor technology: A generic framework," in *International Symposium on Nanoscale Architectures*, July 2016, pp. 43–48.

[10] S. Shirinzadeh, M. Soeken, P. E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *Design, Automation Test in Europe Conference Exhibition*, March 2016, pp. 948–953.

[11] Y. Zha and J. Li, "Reconfigurable in-memory computing with resistive memory crossbar," in *International Conference on Computer-Aided Design*, Nov 2016, pp. 1–8.

[12] S. Kvatinsky, N. Wald, G. Satat, A. Kolodny, U. C. Weiser, and E. G. Friedman, "MRL: Memristor Ratioed Logic," in *International Workshop on Cellular Nanoscale Networks and their Applications*, Aug 2012, pp. 1–6.

[13] L. Guckert and E. E. Swartzlander, "MAD gates: Memristor logic design using driver circuitry," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 64, no. 2, pp. 171–175, Feb 2017.

[14] G. Papandroulidakis, I. Vourkas, N. Vasileiadis, and G. C. Sirakoulis, "Boolean logic operations and computing circuits based on memristors," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 61, no. 12, pp. 972–976, Dec 2014.

[15] G. Papandroulidakis, I. Vourkas, A. Abusleme, G. Sirakoulis, and A. Rubio, "Crossbar-based memristive logic-in-memory architecture," *IEEE Trans. Nanotechnol.*, 2017 (in press).

[16] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[17] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, "Logic operations in memory using a memristive akers array," *Microelectronics Journal*, vol. 45, no. 11, pp. 1429 – 1437, 2014.

[18] E. Lehtonen, J. H. Poikonen, and M. Laiho, "Memristive stateful logic," in *Memristor Networks*, A. Adamatzky and L. Chua, Eds. Springer.

[19] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: the terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, Apr 1995.

[20] S. M. Hassan, S. Yalamanchili, and S. Mukhopadhyay, "Near data processing: Impact and optimization of 3D memory system architecture on the uncore," in *International Symposium on Memory Systems*.

[21] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 481–492.

[22] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise AND and OR in DRAM," *IEEE Comput. Archit. Lett*, vol. 14, no. 2, pp. 127–131, July 2015.

[23] R. Ben-Hur and S. Kvatinsky, "Memristive memory processing unit (MPU) controller for in-memory processing," in *International Conference on the Science of Electrical Engineering (ICSEE)*, Nov 2016, pp. 1–5.

[24] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *International Symposium on High Performance Computer Architecture*, Feb 2015, pp. 476–488.

[25] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono, "An 8 mb multi-layered cross-point reram macro with 443 mb/s write throughput," *IEEE J. Solid-State Circuits*, vol. 48, no. 1, pp. 178–185, Jan 2013.